15-854: Approximations Algorithms **Lecturer:** R. Ravi
Topic: A Primal-Dual Algorithm for Minimum-Cost Steiner Tree **Date:** Nov. 28th, 2005
Scribe: Avi Gavlovski

22.1 A Primal-Dual Algorithm for Minimum-Cost Steiner Tree

Given an undirected graph $G = (V, E)$, a cost function c on the edges $c : E \to \mathbb{Q}^+$, and a set of terminals[1] $T \subseteq V$, we look for a subgraph $H = (V_H, E_H)$ of G connecting the terminals, of minimum cost; where the cost is given by $\sum_{E_H} c_e$.

We will first formulate the problem as an integer linear program, which we then relax. From here we will derive its dual, and the constraints in the dual suggest us an algorithm that yields a $2*(1-\frac{1}{|T|})$ approximation, and is actually similar in structure to running Dijkstra's algorithm gradually and evenly from each terminal.

22.2 The Linear Programming Formulation

Let us introduce the following definitions:

We say that a set $S \subset V$ *separates* T if $S \cap T \neq \emptyset$ and $S \cap T \neq T$.

Denote by $\partial(S)$ the set of edges with exactly one endpoint in S.

The linear program is formulated based on the following observation. Given a set S that separates T, any Steiner tree connecting T must use at least one edge in $\partial(S)$, since otherwise $S \cap T$ will be disconnected from $S^c \cap T$, and by the definition of a separating set both of these subsets of T are nonempty.

For each edge $e \in E$, let variable x_e represent whether the edge is chosen for our subgraph, i.e. $x_e = 1$ iff $e \in E_S$.

$$\begin{array}{lll} \min \sum_{e\in E} c_e * x_e & \text{subject to} & \\ \sum_{e\in\partial(S)} x_e \geq 1 & \forall S \textit{ separating } T & \qquad (\textbf{Primal}) \\ x_e \in \{0,1\} & \forall e \in E & \end{array}$$

By the above observation, any Steiner tree must meet these constraints. Conversely, any subgraph meeting these constraints will connect T. Since a minimum subgraph connecting T must be a tree (otherwise drop an edge from any cycle), we have that this integer linear program gives the optimal Steiner tree connecting T.

We relax this LP dropping the constraints $x_e \in \{0, 1\}$ and only require $x_e \geq 0$.

[1]Note that for $|T| = 2$ this is the shortest path problem.

22.3 The Dual

The dual[2] of the above LP is:

$$\begin{array}{lll} \max \sum_{\{S \textit{ separating } T\}} y_S & \text{subject to} & \\ \sum_{e\in\partial(S)} y_S \le c_e & \forall e \in E & (\textbf{Dual}) \\ y_S \in \{0,1\} & \forall S \textit{ separating } T & \end{array}$$

Can we give an interpretation to this ILP? In the previous lecture we saw that the dual of a linear programming formulation for Vertex Cover was an ILP formulation for Maximum Matching (given a constant cost function).

Assume $c_e = 1$ for all edges. Since each set S separates T in the graph, we are looking for a maximum collection of separating sets. However the constraint $\sum_{e\in\partial(S)} y_S \le c_e = 1$ tells us that we only consider edge-disjoint separating sets. Hence we are simply looking for a largest collection of edge-disjoint cuts.

We relax this LP dropping the constraints $x_e \in \{0,1\}$ and only require $y_S \ge 0$.

22.4 Motivation for the algorithm

By complementary slackness conditions we are interested in having some or all of the constraints be tight. Hence we can imagine growing some of the y_S's evenly until one of the constraints is tight (hereon *complains*). Then at this point we could choose the edge e corresponding to the complaining constraint, and for all S containing edge e we fix y_S at its current value. We then could keep going and pick some more edges when the corresponding constraint complains, until we're done.

However, initially the easiest way to separate T is separating one of the terminals from the rest...

22.5 The algorithm

We initialize a set of components $\mathcal{C} = \{(M_C(x), T_C(x)) : x \in T\}$, where we $M_C(x) = \{y_{\{x\}}\}$ and $T_C(x) = (\{x\}, \emptyset)$ for each terminal x. That is, T_C is initialized to the tree consisting of terminal x and no edges. Also, M_C, referred to as the *moat* of the component, will always be a set of dual variables y_S such that $x \in S$ for each terminal x in C (later on the components will contain more than one terminal, as we will see). It will always be the case that $C \in M_C$.

Throughout the algorithm we maintain the invariant that if $C \in \mathcal{C}$, then the vertices of T_C separate T.

[2]See lecture 21.

Initialize a forest $F = (T, \emptyset)$. At the end of the algorithm F will be our Steiner tree. Another invariant will be that the forest F will always be a subgraph of the union of the T_C's, and that the restriction of F to a particular C is a tree.

Initialize all dual variables $y_S = 0$ and initialize a "time" variable $t = 0$.

We gradually increase t, and for all $C \in \mathcal{C}$ we increase y_C at the same rate as t. That is, for every time interval $(t, t + \Delta t)$ in which we don't alter $\mathcal{C}$, we increase each y_C by Δt. Notice that in any time interval we are growing exactly one of the variables in the moat of each component C, namely y_C. And $M_C \setminus \{y_C\}$ is the nothing but the set of variables that we have grown in the "history" of C, which we will later use for our analysis.

Every once in a while, at time t one of the constraints $\sum_{e \in \partial(S)} y_S \leq c_e$ becomes tight for some edge e. By construction, we know that one of the endpoints of this edge is in one of the components C_i of $\mathcal{C}$. Let u be this endpoint and let v be the endpoint which is not in C_i.

Case 1: v belongs to some other component C_j. Then we combine C_i and C_j into a new component C'. We set $T_{C'} = T_{C_i} \cup T_{C_j} + e$. That is, we combine the two trees into a larger tree. By invariant, the restriction of F to each of these two components is a tree, so the combined tree $T_{C'}$ contains a path p connecting $F \cap C_i$ and $F \cap C_j$. We add p (its vertices and edges) to our forest F, which now has one less component (and so does $\mathcal{C}$).

We also combine the moats by setting $M_{C'} = M_{C_i} \cup M_{C_j}$

Case 2: v does not belong to any component in $\mathcal{C}$. Then we simply add v to $V(T_C)$ and e to $E(T_C)$. We also add variable $y_{C \cup \{v\}}$ to M_{C_i}

We stop when there is only one component. By construction, at this point F is a tree connecting all the terminals. The moat of each component keeps track of all the dual variables that we have increased for that component. So the moat of the one component left at the end contains all the dual variables which have been charged throughout the algorithm.

22.6 Performance analysis

Define F_C to be the restriction of F to component C, and

define $y(C) = \sum_{S \in M_C} y_S$

and $cost(C) = \sum_{e \in E(F_C)} c_e$

Claim: At any time t, for any component C we have $cost(C) \leq 2(y(C) - t)$

This holds at $t = 0$ since $y(C) = cost(C) = 0$ for any component C.

Now we assume that it holds at time t, and show that it holds at time $t + \Delta t$ for some $\Delta t > 0$.

Case 1: There is a collision at time t (case 1 of the algorithm).

Let C_1 and C_2 be the two components that collide.

Then by construction $y(C) = y(C_1) + y(C_2)$
and $cost(C) = cost(C_1) + cost(C_2) + 2t$

But by the inductive hypothesis we have:
$cost(C_1) \leq 2(y(C_1) - t)$ $cost(C_2) \leq 2(y(C_2) - t)$
so $cost(C) = cost(C_1) + cost(C_2) + 2t \leq 2(y(C_1) + y(C_2) - t) = 2(y(C) - t)$
Case 2: There is *no* collision in time interval [t, t + Δt) (case 2 of the algorithm).

Then by construction $y_{new}(C) = y_{old}(C) + \Delta t$
and $cost_{new}(C) = cost_{old}(C)$

But by the inductive hypothesis we have $cost_{old}(C) \leq 2(y_{old}(C) - t)$ so

$cost_{new}(C) = cost_{old}(C) \leq 2(y_{old}(C) + \Delta t - (t + \Delta t)) = 2(y_{new}(C) - (t + \Delta t))$ ∎(Claim)

We can think of each of the components as a company making money. Each company always has $2t$ money in its bank account, where t is the time since the company was founded, and all of these companies are founded at the same time.

Companies make money Δt for every interval of time Δt that goes by, unless there is a merge. Every once in a while, at some time t, two companies will merge. Merging costs them $2t$, and since each of them has then a savings of $2t$, they combined have $4t$ money, so they spend $2t$ in the merge and still have $2t$ to put in the bank.

For the final analysis, note that $y_{final}(\mathcal{C}) \leq |T| * t_{final}$ since there are at most $|T|$ components at any point in the course of the algorithm, so for every interval of time of length Δt, the total value being added to the moats is at most $|T| * \Delta t$.

Hence by this observation we have:

$$\begin{aligned}
cost_{final}(\mathcal{C}) &\leq 2(y_{final}(C) - t_{final}) \\
&\leq 2(y_{final}(C) - \frac{y_{final}(C)}{|T|}) \\
&\leq 2(1 - \frac{1}{|T|})y_{final}(C)
\end{aligned}$$

so we have a $2(1 - \frac{1}{|T|})$-approximation. ∎